Temporal Geo-Social Personalized Search Over Streaming Data*

Abdulaziz Almaslukh[a,b] Amr Magdy[a,b]
[a]Department of Computer Science and Engineering, [b]Center for Geospatial Sciences
University of California, Riverside
aalma021@ucr.edu,amr@cs.ucr.edu

ABSTRACT

The unprecedented rise of social media platforms, combined with location-aware technologies, has led to continuously producing a significant amount of geo-social data that flows as a user-generated data stream. This data has been exploited in several important use cases in various application domains. This paper supports geo-social personalized queries in streaming data environments that have not been addressed in the existing literature. We define two temporal geo-social queries that provide users with real-time personalized answers based on their social graph. Then, we propose an indexing framework that provides lightweight and effective real-time indexing to digest geo-social data in real time. The framework distinguishes highly-dynamic data from relatively-stable data and uses appropriate data structures and storage tier for each. Based on this framework, we propose a novel geo-social index and adopt two baseline indexes to support the addressed queries. The query processor then employs different types of pruning to efficiently access the index content and provide real-time query response. The extensive experimental evaluation based on real datasets has shown the superiority of our proposed techniques to index real-time data and provide low-latency queries compared to existing competitors.

CCS CONCEPTS

• **Information systems** → **Multidimensional range search**; **Stream management**; **Data streaming**; *Query operators.*

KEYWORDS

Spatial, Temporal, Geo-social, Real-time, Indexing, Query Processing

ACM Reference Format:
Abdulaziz Almaslukh[a,b] Amr Magdy[a,b]. 2019. Temporal Geo-Social Personalized Search Over Streaming Data. In *27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19), November 5–8, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3347146.3359073

*This work is partially supported by the National Science Foundation, USA, under grants IIS-1849971, SES-1831615, and CNS-1837577.

SIGSPATIAL '19, November 5–8, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-6909-1/19/11…$15.00
https://doi.org/10.1145/3347146.3359073

1 INTRODUCTION

The unprecedented popularity of online social media platforms over the past decade combined with the availability of location information through GPS-equipped devices has led to significant attention for supporting geo-social queries at scale [5, 6, 13] in order to serve applications efficiently on such big data. These queries are used in various applications and services such as social recommendations [8, 35, 44], community and event detection [10, 21, 41], and urban planning [17]. A major category of these queries is personalized search queries that use the social information to tailor the query answer per the issuing user. For example, a user who is visiting *Paris* wants to find recent posts from her friends in the city. To allow finding recent posts at a fine temporal granularity, it is required to manage geo-social data as a data stream. In fact, the modern geo-social data has a streaming nature due to the large number of its data items that arrive every second around the clock. Latest assessments estimate Twitter to receive approximately 8,500 tweets/second [19] while Facebook posts are even an order of magnitude larger in size [14, 19]. This streaming nature has already motivated several streaming queries on this data, such as keyword queries [3, 26, 37], spatial queries [25, 28], and social queries [23, 30], with plenty of applications. Although several geo-social queries have been addressed in the literature [5, 6, 13, 20, 22, 39, 45], querying streaming data combining both social and geographic location information is still an unaddressed challenge.

Geo-social queries have got a little attention in the streaming environments although several applications that are powered by these queries will significantly benefit from the real-time nature of geo-social data, e.g., providing real-time search on friends' posts during emergency situations and detecting real-time events based on friends' updates. In such streaming environments, hundreds of millions of items arrive at high pace every day, which puts major challenges on real-time indexing and query processing based on both social and geographic information. These challenges include sustainable digestion of new data in real-time index structures and exploiting the social information, which is usually complex in structure and huge in size, to serve incoming queries that have certain locations of interest. State-of-the-art techniques [7, 22, 33, 34] are still limited to address these challenges, either for inefficient indexing for real-time data or inefficient query processing navigating highly-complex graph structures, which limits using streaming geo-social information in scalable applications.

This paper introduces scalable real-time indexing and query processing for two geo-social personalized search queries over streaming data. The two queries combine three aspects: spatial, temporal, and the social connectivity between users. They are socio-temporal extensions of the two fundamental spatial queries, range query and k-nearest-neighbor query, to effectively serve the streaming data

applications that are timely by nature. Example of such queries is to *"find what my friends/friends-of-friends have recently posted in Paris"*, where a spatial range encapsulates *Paris* city boundaries, or *"find what my friends/friends-of-friends post now nearby Tampa, Florida"* in case of hurricane emergency. Such queries are obviously useful for various applications that make use of personalized real-time content, such as improving emergency response by involving the close social circle of individuals or getting personalized recommendations from friends. To limit query answer to top relevant items, the queries use ranking functions based on timestamp and discrete social distance, similar in spirit to hop count, to retrieve only top-k items that satisfy the query predicates.

In support of these two queries in real time, we propose a geo-social indexing framework that distinguishes highly-streaming data from relatively-stable data. Then, it employs memory-based light indexing for incoming streams and disk-based indexing for stable data. Based on this framework, we propose a novel geo-social index that effectively organizes real-time data for efficient querying. The index consists of three components: an in-memory spatio-temporal index, an in-disk social index, and an in-memory buffer. During query processing, both in-memory and in-disk data are combined to retrieve relevant data from direct friends in the social graph. If the retrieved data items are less than k, then the query search expands to search indirect friends at one or more levels of social expansions to retrieve the final top-k answer. Due to the awareness of social aspect, the query processor smartly prunes the search space based on social connectivity in addition to spatial and temporal information. Such three-dimensional pruning significantly reduces the query response time and reduces contention on the real-time index structure to maintain high real-time data digestion rates.

The extensive experimental evaluation of our proposed techniques on real datasets has shown superiority over competitor techniques that are incorporated from the literature. Using a single machine setting, our index can digest up to 220K object/second of streaming data while providing an order of milli-seconds query latency for both average and 99% of the queries. In addition, the in-memory component of our proposed index consistently maintains low memory usage compared to competitor techniques. Our contributions in this paper can be summarized as follows:

- We define two temporal geo-social personalized search queries that retrieve data objects based on spatial, temporal, and social predicates on streaming data.
- We propose a novel real-time indexing framework that efficiently digests geo-social streaming data in real time.
- We develop query processing techniques that exploit the index content and further prune the search space to provide low query latency.
- We extensively evaluate the proposed techniques compared to existing competitors on real Twitter datasets showing their superiority and effectiveness for streaming environments.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 presents the problem definition. Sections 4 and 5 detail the proposed geo-social indexing and query processing techniques. Section 6 provides an extensive experimental evaluation. Finally, Section 7 concludes the paper.

2 RELATED WORK

There is no current research work that addresses geo-social queries on user-generated streaming data in real time to the best of our knowledge. However, social-aware queries are supported independently on both spatial user-generated data and streaming user-generated data in the literature. This section covers this literature and distinguishes it from our proposed work.

Queries on user-generated streaming data. User-generated streaming data has got significant attention over the past few years due to the popularity of online social media platforms and similar online services. In addition to continuous queries [29, 37] that was the only focus of traditional machine-generated streaming data, user-generated streaming data has been exploited for various applications and snapshot queries, such as geo-textual queries [3, 11, 26], location-based search [7, 9, 28], trend detection [1, 15, 31], time-sensitive recommendations [42], and news and topic extraction [16, 32, 38]. In this literature, the spatial and social aspects of the queries are addressed independently. So, location-based search queries, e.g., [7, 9, 28], do not support any social or personalized aspect, and personalized queries, e.g., [22], do not consider the spatial dimension. A recent attempt to combine both spatial and social dimension is proposed in [33]. However, their solution creates a complete disk-based spatial index for each user, which is extremely expensive for streaming data and cannot even scale to be a baseline approach to compare with. Our work distinguishes itself from existing techniques to be the first to combine both spatial and social aspects in one query while considering streaming environments for both lightweight real-time indexing and efficient query processing.

Social queries on spatial data. Due to the importance and various applications that benefit from combining social and spatial aspects, several researchers have recently developed indexing and query processing techniques for different geo-social queries, e.g., [2, 5, 6, 13, 30, 45]. This includes recommending POIs [34, 35, 43, 44], finding cliques [18, 24, 40], finding top-k spatial-keyword objects [39], and finding top-k influential users [20]. However, none of these techniques address geo-social personalized search queries on streaming data. Thus, our work is distinguished from all existing techniques in multiple ways. First, we are the first to extend geo-social queries with the temporal aspect due to the nature of streaming data that is the main focus of this paper. Second, we are the first to consider lightweight real-time indexing and query processing for geo-social data. This real-time aspect of the streaming environment puts significant overhead on both indexing and query processing, which cannot be handled by any of the existing techniques.

3 PROBLEM DEFINITION

We evaluate the geo-social queries on a streaming dataset D that consists of geo-social objects. Each object $o \in D$ is represented with the three main attributes (uid, loc, $timestamp$), where uid is the identifier of user who posted this object, loc is the location of the object in the two-dimensional space represented with latitude/longitude coordinates, and $timestamp$ is the time when the object is posted. D_T is a snapshot of the dataset D at time T, so every object $o \in D_T$ has $o.timestamp \leq T$. Table 1 shows a sample

UID	OID	Keywords	Timestamp
$u1$	$o1$	Fantastic, Comeback, Play	06-10-2019 20:18:30
$u2$	$o2$	Love, Pineapple, Pizza	06-10-2019 20:18:27
$u3$	$o3$	Sunny, Day, Good, Running	06-10-2019 20:18:23
$u1$	$o4$	Freeway, Traffic, Bad	06-10-2019 20:18:19
$u4$	$o5$	University, Graduation	06-10-2019 20:18:17
$u2$	$o6$	USA, Japan, Summit	06-10-2019 20:18:14
$u5$	$o7$	Airport, Flight, Time, Ready	06-10-2019 20:18:09
$u6$	$o8$	NBA, Lakers, LeBron	06-10-2019 20:18:06

Table 1: Content of Objects in Figure 1

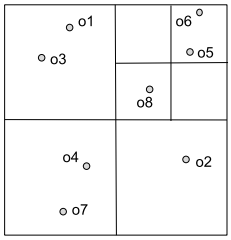


Figure 1: Spatial Quadtree (SQ)

of the dataset that consists of eight objects. Each object, identified by *oid*, is composed of a user id who posted the object, a set of keywords that represent the textual content, a timestamp, and located in the space as shown in Figure 1. In addition, the social connectivity between the users is represented as a hashtable where the <key,value> pair is <user id, list of friend ids>. Each entry consists of the given user id as the key, and the list of user's friends ids as the value. We can easily navigate from user's friend to the friends of friends by expanding the immediate friends and retrieving their friends. This process can be repeated to navigate to higher levels of the social graph. The simplicity of representation and navigation of the social graph helps the query processors to achieve high query throughput, especially in a tight streaming environment.

The two fundamental spatial queries, in particular range query and k-nearest neighbor, that are common in the literature have been extended to support temporal geo-social aspects in this work. The query definitions of the two extended queries as following:

Spatial-social Temporal Range Query (SSTRQ): given q = <user u, spatial range R, integer k, and timestamp T>, and D_T that is a snapshot of the dataset D at time T, SSTRQ retrieves the most recent k objects $o_i \in D_T$, $1 \leq i \leq k$, that are posted within R and are posted by u's friends or friends of friends based on a discrete social distance.

The k objects are ranked based on time to retrieve the most recent objects in D_T from u's direct friends. If q fails to retrieve all k objects from u's friends, the search is expanded to u's friends of friends recursively to retrieve the rest of objects. So, the social relevance of objects in q answer are assessed based on a discrete social distance that takes only integer values (1,2,3, etc) and no fractional values in between. This enables scalable query processing on streaming data in real time as detailed in the following sections.

Spatial-social Temporal kNN Query (SSTkQ): given q = <user u, spatial point location L, integer k, and timestamp T>, and D_T that is a snapshot of the dataset D at time T, SSTkQ retrieves top-k objects $o_i \in D_T$, $1 \leq i \leq k$, that are posted by u's friends or friends of friends, and ranked based on a spatio-temporal distance F_α from L and T as follows:

$$F_\alpha(o, q) = \alpha \times SpatialScore(o, q) + (1 - \alpha) \times TemporalScore(o, q)$$

Where α is a weighting parameter, $0 \leq \alpha \leq 1$, that weights the relative importance of spatial and temporal scores in the object proximity. $SpatialScore$ and $TemporalScore$ are defined as follows:

$$SpatialScore(o, q) = \frac{distance(o.loc, q.L)}{R_{Max}}$$

$$TemporalScore(o, q) = \frac{q.T - o.timestamp}{T_{Max}}$$

Where R_{Max} and T_{Max} are the maximum allowed spatial and temporal ranges for any object, and *distance* is the spatial distance between object and query locations in the Euclidean space. The social relevance is assessed using the same discrete social distance that is used in SSTRQ for scalability on streaming data in real time.

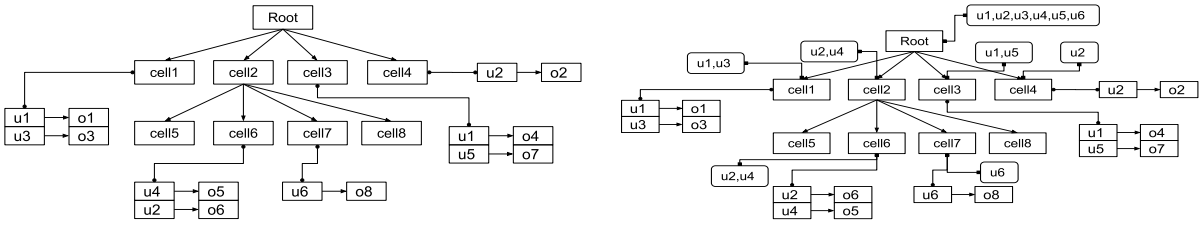
4 GEO-SOCIAL REAL-TIME INDEXING

This section presents geo-social data indexing in real time. This data is rich with spatial, temporal, and social information. The two main challenges in indexing such rich data in real time are: (1) encoding the incoming information in highly-scalable data structures that are efficient for insertions with tens of thousands of data objects each second, and (2) removing old data from the main memory to sustain digesting new incoming data objects at all times. Traditional insertion procedures in spatial and social index structures incur significant overhead that limits scalable data digestion. In addition, straight forward deletion procedure that scan every index cell in different spatial regions or different parts of the social graph to expel old data incur significant overhead that will also affect the indexing scalability in real time.

To address these challenges, we introduce a generic indexing framework (Section 4.1) that separates highly-dynamic data from relatively-stable data, so real-time data structures are tailored to digest only the needed information in real time to reduce both insertion and deletion overheads. Based on this framework, we propose a scalable index (Section 4.2) that enables efficient handling for geo-social data in real time, and adapt two baseline index structures (Section 4.3) from the literature of spatial and spatial-social indexing. The rest of this section details the indexing framework as well as the three indexes.

4.1 Indexing Framework

The proposed indexing framework depends on the observation that incoming geo-social data objects are highly dynamic while the social graph information is relatively static. Each second, tens of thousands of geo-social objects are flowing, which requires real-time digestion. These objects are posted by hundreds of millions of users that are connected to each other with social bonds, represented as a social graph. This social graph is not updated frequently compared to the geo-social objects. In real Twitter dataset, an active user posts on average seven tweets per day [36], which leads to hundreds of millions of tweets every day. However, the number of



(a) Spatial-social Quadtree (SSQ) **(b) Tightly-coupled Spatial-social Quadtree (TCSSQ)**

Figure 2: Structure of geo-social real-time indexes

new friends or unfollowed friends are not even close to this daily number. It is usual not to accept new friends or follow new people for several days, weeks, or even months. Consequently, the frequency of updates in social graph information is way less than the incoming geo-social objects in real time. Our indexing framework exploits this observation to dedicate the necessary resources to index each type of data.

The proposed indexing framework consists of three components: (1) in-memory index that digests streaming geo-social objects in real time, (2) in-disk index that organizes relatively stable social graph information, and (3) in-memory buffer that swaps social graph information from and to the disk index. The in-memory index is equipped with optimized insertion and deletion techniques that minimize the real-time overhead and enable to scale for handling streaming data. As main-memory is a scarce resource, data cannot be digested infinitely with excessive amounts and have to be expelled to a secondary storage on a regular basis. For that reason, the in-memory index employs a temporal duration T_{Max} that indicates the maximum allowed past data to store. T_{Max} is a system parameter and can be adjusted by the administrators based on the available main-memory resources and the streaming rates of incoming data.

The second component is an in-disk index that stores the social graph information. Two reasons are behind storing this information on disk. First, the excessive size of this information consumes significant memory storage that is not frequently utilized, due to the long-tail distribution where the majority of users are inactive in queries [27]. For example, a subset of our experimental Twitter social graph with 3.3 million users consumes approximately 62.5 GB of main-memory as each user has an average of 500 friends. Second, the relative stability of social graph information as discussed earlier in this section. This makes the social graph index structure needs infrequent updates, which is not challenging to be handled on the disk storage. However, for query processing, it is inefficient to visit the disk for every retrieval of a user friend list, especially for active users who post frequent queries. This has motivated the third component of our indexing framework, which is the in-memory buffer for social graph information. This component acts similar to the database buffer, where certain disk pages are swapped in the main-memory buffer from the disk index only when needed. As disk pages keep accumulating in the buffer, it becomes full and needs to evict some of its content to swap in new pages. Eviction policies that are used for the buffer are the same ones studies in the literature of database buffer management and operating system virtual memory. We choose to use the famous least recently used (LRU) policy in our realization. However, other policies could be used based on the underlying application requirements.

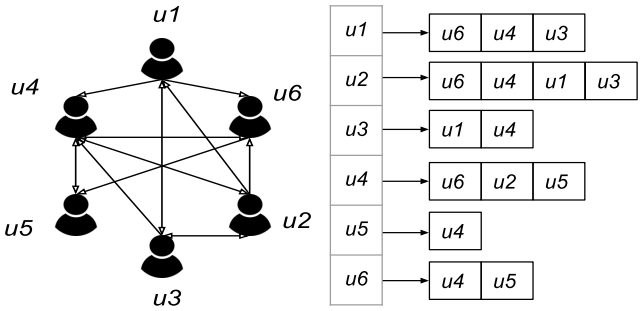
4.2 SSQ Index

Based on the described framework in Section 4.1, we propose *Spatial-Social Quadtree (SSQ)* index for scalable real-time indexing of geo-social objects. Conformed to the framework, the index has three components, an in-memory component for digesting objects in real-time, a disk-resident component for the social graph indexing, and an in-memory buffer, as described in Section 4.1. This section describes the details of index structures and update operations for different index components.

Index structure. The in-memory component adopts a spatial quadtree [4] as a highly-scalable space-partitioning index for real-time data digestion. An example of spatial quadtree is depicted in Figure 1 for eight geo-social objects that are presented in Table 1. The tree divides the space into multi-level disjoint cells that either have four or zero children cells. An incoming object is located in the cell that contains its location. A cell is divided into four quadrants only if the number of objects exceeds a specific cell capacity, which is a system parameter that determines the tree height, so a small cell capacity leads to a deeper tree while a large cell capacity generates a shallow tree. Only leaf nodes hold data objects, while intermediate nodes provide routing information. SSQ index extends the quadtree to be aware of the user aspect of the spatial objects. In specific, each leaf cell is equipped with a hash index structure that organizes the cell's objects based on the issuing users. This hash structure is light for real-time digestion, and still provides effective pruning for the search space based on the social information. The hash structure uses the user id as a key and the value is a list of objects that are posted by this user ordered based on their timestamps. Including the social information within the spatial cell significantly helps the query processor to retrieve candidate objects that could potentially make it to the final answer.

Figure 2a depicts an example of the SSQ in-memory index. The depicted index represents the same set of objects that are depicted in Figure 1, and the same quadtree organization, with adding the light hash structure to each leaf node that enables effective social-based pruning while sustains high digestion rates in real time as verified in our experimental evaluation.

The in-disk component of SSQ index stores the social graph represented by a set of adjacency lists. Our social graph representation adopts the famous form that represents users as nodes and friendship relations as directed edges. The adjacency list representation stores this information as a hash structure that uses user id as a key and list of friends as a value for each hash entry. Figure 3



(a) Example Social Network **(b) Social Structure on Disk**

Figure 3: Example of In-disk Social Structure

demonstrates an example for a social graph with six users, $u1$ to $u6$. Figure 3a shows the high-level graph model for the social relations among the six users while Figure 3b shows the adjacency list representation that is stored on the disk-resident index structure. The disk structure consists of two parts, the data part and the index part. The data part stores consecutive blocks of long integer lists that contain the user ids as depicted in Figure 3b. The index part stores all the distinct user ids, each user id is associated with a disk pointer to the block in which the user friend list is stored. Compared to the data part, the index part is small in size and can be easily loaded during the query processing for efficient access of user information as described in Section 5.

To reduce the overhead of reading back and forth from the disk, the third component of SSQ index is a dedicated in-memory buffer that is utilized to store the retrieved user friend lists from the disk for further recycling during future queries. The in-memory buffer is a hash structure that stores key-value pairs of user ids and friend lists, similar in format to the disk index from which data is retrieved. When the in-memory buffer is full, it adopts the least recently used (LRU) policy to free up content to continue serving incoming queries.

Index insertion. Insertion in both the in-disk component of SSQ index and its corresponding buffer adopts traditional one-by-one insertion due to the low insertion rates in the stable social structure. On the contrary, the in-memory index component, that adopts a social-aware quadtree, incurs an excessive insertion rate as tens of thousands of objects arrive every second. Traditional insertion procedure that navigates the tree hierarchy for each incoming object and inserts it in the corresponding cell does not scale to cope up with such high insertion rate. To overcome this problem, we employ a batch insertion process that collects a few seconds worth of data in a temporary buffer and inserts them as one batch in the quadtree structure. During the buffering, a minimum bounding rectangle (MBR) is maintained around the location of incoming objects. Then, the MBR boundaries are compared to the index cell boundaries, instead of comparing location of each object, and the tree navigation is performed based on this cheap comparison. With thousands of objects buffered, thousands of comparison operations are saved, which significantly boost the digestion performance and allows to ingest streaming data with high arrival rates.

Index deletion. To sustain digesting incoming data in the scarce memory resources, the in-memory index expels objects that are older than T_{Max} time units ago to the disk, where T_{Max} is a system parameter that is based on the availability of memory resources and arrival rates of the underlying streaming data. To expel this data, a straight forward way is to exhaustively iterate over all index cells, either every few time units or when a certain memory budget fills up, and clean up all expired data objects that are older than T_{Max}. However, such exhaustive and frequent cleaning process puts an overhead on real-time operations of the index. To avoid such overhead, we employ a combination of regular and periodic cleaning processes that are lighter than the exhaustive cleaning and still sustain memory consumption. The regular cleaning is piggybacked on the real-time insertion and querying, so whenever an index cell is accessed for either insertion or query processing, the accessed entries are checked for expired content to be expelled from main-memory. This reduces the cleaning overhead as it shares the index traversal overhead with the other operations.

This regular cleaning process does not guarantee to expel all the expired data proactively as it depends on the spatial distributions of both data and queries, so some index cells might be left without cleaning due to infrequent access to those cells. To address this, we employ a light periodic cleaning that goes over all index cells every T_{Max} time units. For each cell, if it is not cleaned during the past T_{Max} time units, which means no insertions happened during this period, all the cell content is wiped as all objects are expired. Otherwise, the cell is skipped. This process is very light and mainly addresses cells that are infrequently accessed. In addition, it can be easily invoked in a separate thread to reduce the contention over index cells in real time.

4.3 Baseline Indexes

In addition to our proposed SSQ index (Section 4.2), we adopt two baseline indexes based on the proposed indexing framework that is described in Section 4.1. The two baseline indexes are alternatives to address the supported queries based on existing techniques in the literature. The two baseline indexes are *Spatial Quadtree (SQ)* and *Tightly-Coupled Spatial-Social Quadtree (TCSSQ)*. The rest of this section describes each index and highlight its differences compared to the proposed SSQ index.

(1) *Spatial Quadtree (SQ)*. This index has a similar structure to the SSQ index with the exception of the in-memory index component that adopts a pure spatial quadtree structure without any extended structures to organize the data based on the posting users. Figure 1 shows an example of the spatial quadtree index. It is worth noting that all data objects in the leaf nodes are sorted based on their arrival timestamp at no additional cost due to the nature of streaming data that comes ordered by time. For the index insertion and deletion, the same procedures that are developed for SSQ index are used in SQ index with the exception of navigation the leaf nodes content that does not have the hash structure anymore. So, inserted data are appended to a long list of chronologically ordered objects, and all the cleaning processes are performed on the same list, which reduces the real-time indexing overhead while increases the query processing overhead as will be detailed in Sections 5 and 6.

(2) *Tightly-Coupled Spatial-Social Quadtree (TCSSQ)*. This index has a similar structure to the SSQ index with the exception of the in-memory index component that includes extra user information in all intermediate and leaf nodes of the quadtree structure

instead of having a hash structure in only leaf nodes. In specific, each leaf node C has an additional list of users $C.L_u$ who posted in the spatial region of C. Then, the content of $C.L_u$ is replicated to the parent nodes up to the root node. So, the root's L_u has all the users who posted in any region, and each intermediate node has a list of all users who posted in the sub-tree that is rooted in this intermediate node. This organization is a modified version of [39] that is suitable for real-time indexing. This is built based on the core ideas of the IR-tree structure [12]. Figure 2b depicts an example of TCSSQ index for the eight objects of Table 1. Each node, including root, intermediate, and leaf nodes, has an additional list $C.L_u$ of users who posted in the node C spatial region.

The additional user lists L_u affect the index insertions and deletions in real time. On insertion, after the insertion procedure is performed in node C as described for SSQ index, the posting user id *uid* is added to $C.L_u$. To this end, *uid* is searched in $C.L_u$ using binary search. If *uid* does not exist in $C.L_u$, it is inserted into the ordered list, otherwise, $C.L_u$ remains intact. Then, the same process repeats for parent nodes' L_u until it propagates to the root node. On index deletion, object deletions are performed for certain user entries in the node's hash structure. For each user entry, if the list of objects remains non-empty, i.e., there are still remaining objects for this user in the node, $C.L_u$ remains intact. On the contrary, if the list of objects becomes empty, i.e., the deleted objects are the last objects for this user in the node, then the user id is removed from $C.L_u$. Then, the removal checks are propagated to parent levels of the tree. For C's parent L_u, the three siblings nodes of C are checked. If *uid* exists in any of their L_u lists, then the parent's L_u remains intact. If *uid* does not exist in any of these lists, then *uid* is removed from the parent's L_u, and the removal check is propagated to the higher levels up to the root node.

5 QUERY PROCESSING

This section details the query processing of SSTRQ and SSTkQ queries that are defined in Section 3 exploiting the proposed SSQ index, and the baseline SQ and TCSSQ indexes that are introduced in Section 4. In Section 5.1, we introduce a high-level query processing framework that is generic for the three indexes. Then, Sections 5.2 and 5.3 detail the query processing of SSTRQ and SSTkQ queries, respectively, based on the three indexes.

5.1 Query Processing Framework

Our query processor consists of two generic steps:

(1) Step 1: Given the user id *uid* of the query issuing user u, step 1 retrieves a list of friends $u.L_f$ that contains a set of user ids for u's direct friends. To this end, the in-memory buffer of the social graph is checked with the key value *uid*. If it exists, $u.L_f$ is directly retrieved from the buffer. Otherwise, the in-disk social index is accessed in a traditional way to retrieve $u.L_f$ to the in-memory buffer. If the in-memory buffer is full, the least recently used (LRU) replacement policy is used to free up some of the buffer content. Then, $u.L_f$ is fed to step 2 of the query processor.

(2) Step 2: Given a list of friends L_f, that is retrieved in step 1, and spatio-temporal predicates, in step 2, the query processor accesses the in-memory spatial index to retrieve the top-k objects based on the query semantic and the underlying index structure. The specifics of this step is different for each <query,index> combination, as detailed in the rest of this section.

If the execution of these two steps retrieves k objects, then they are considered a final query answer and returned to the user. If the computed answer has less than k, the search is expanded recursively beyond u's social level 1 (direct friends) to social level 2 (friends of friends) or higher social levels until k objects are retrieved. To this end, the two steps are repeated for each user id in L_f for expansion to social level 2, and the same repeats for higher social levels.

5.2 SSTRQ Query Processing

This section details the specifics of step 2 of Section 5.1 for SSTRQ query. In this step, the query processor retrieves the most recent k objects within a spatial region R, per the query definition, that are posted by users in the friend list L_f that is computed in step 1. The rest of this section details this procedure using SSQ, SQ, and TCSSQ indexes.

SSTRQ in SSQ index. SSTRQ query is processed on three phases in SSQ index: (a) spatial retrieval, (b) social filtering, and (c) temporal pruning. First, the spatial retrieval phase navigates the quadtree to retrieve the tree nodes that intersect with the query region R. Second, for each node, the social filtering phase accesses the hash index and retrieve lists of objects that are associated with user ids in the friend list L_f. Each of these lists is ordered based on timestamp due to the streaming nature of incoming objects. Third, the retrieved lists are enqueued in a priority queue Q that orders lists based on their most recent object. Then, the lists are traversed in Q order to compute an initial answers *Ans* of k objects. Based on *Ans*, a temporal boundary T_k is computed as the timestamp of the k^{th} object in *Ans*. Any object older than T_k cannot be part of the final answer. So, T_k is used as a temporal pruning boundary to process the rest of the objects in Q. In specific, each list in Q is retrieved in order. Then, the list's objects are traversed in time order. If the current object $o.timestamp < T_k$, then o is added to *Ans* replacing the k^{th} object, and T_k is updated. Otherwise, o is skipped. Once we reach an object $o.timestamp \geq T_k$, the rest of the list is pruned as no more objects can make it to the final answer. This repeats for all lists in Q before *Ans* is returned as a final query answer.

SSTRQ in SQ index. In SQ index, SSTRQ is processed using the first and third phases, spatial retrieval and temporal pruning, that are used in SSQ index. As SQ index does not include any user information, the social filtering phase cannot be employed. So, the list of objects in each quadtree node is fed directly to the temporal pruning phase that produces the final answer using the same procedure that is described above.

SSTRQ in TCSSQ index. In TCSSQ index, SSTRQ is processed using the same three phases that are used in SSQ index, with an extended social filtering phase. In particular, TCSSQ index maintains extra user list information $C.L_u$ in each quadtree node C. So, the social filtering phase goes through two stages. The first stage is intersecting the user friend list $u.L_f$ with the node user list $C.L_u$. If the intersection is empty, then C and all its descendants are immediately pruned. Otherwise, C is considered for the second stage that is exactly similar to the social filtering phase in SSQ index. The

other two phases, spatial retrieval, and temporal pruning, remains identical to the ones in SSQ index.

5.3 SSTkQ Query Processing

This section details step 2 of the query processing framework that is presented in Section 5.1 for SSTkQ query. This query retrieves the closest k objects, based on a spatio-temporal distance function F_α, nearby a point location L and relative to a query timestamp T that are posted by users in the friend list L_f that is computed in step 1, per the query definition in Section 3. The rest of this section details the query processing using SSQ, SQ, and TCSSQ indexes.

SSTkQ in SSQ index. SSTkQ query is processed on two phases in SSQ index: (a) computing initial answer, and (b) answer refinement. The first phase navigates the quadtree structure to the tree node C that contains the query location L. Then, initial k objects that are associated with users in the friend list L_f are retrieved as an initial answer *Ans*. If C has less than k objects posted by L_f users, then neighbor nodes are checked until *Ans* has k objects.

The second phase uses the k^{th} F_α score of the initial answer (namely $F_{\alpha,k}$) as a refinement boundary to compute the final answer *Ans* so any object with $F_\alpha \geq F_{\alpha,k}$ cannot make it to the final answer. This could be done in a traditional way by visiting all nodes within the maximum spatial range R_{Max} and check objects that are associated with L_f. However, with excessive amounts of data, this could be very expensive and has high query latency. To compute the final answer efficiently, a spatio-temporal pruning procedure is employed to significantly reduce the number of checked objects. To this end, two pruning boundaries are calculated and updated throughout the second phase based on the equation of F_α: a spatial boundary R_u and a temporal boundary T_u. The spatial upper bound R_u is calculated by assuming zero temporal score in the spatio-temporal ranking function, so $R_u = \frac{F_{\alpha,k}}{\alpha} \times R_{max}$. Similarly, the temporal upper bound T_u is calculated by assuming zero spatial score in spatio-temporal ranking function, so $T_u = q.time - \frac{F_{\alpha,k}}{1-\alpha} \times T_{max}$. Any object or cell that are outside R_u and T_u can be safely pruned. So, neighbor quadtree nodes to location L are visited in spatial order with R_u, and objects of each node are checked as long as within T_u. With each new object added to *Ans*, $F_{\alpha,k}$ is updated and then R_u and T_u are updated accordingly. So, the pruning boundaries are continuously tightened, which reduces the total number of checked objects and significantly reduces the query latency. When all nodes and objects within R_u and T_u are exhausted, *Ans* is returned as a final answer.

SSTkQ in SQ index. In SQ index, SSTkQ is processed using the same two phases as in SSQ index with exception to user filtering in quadtree nodes. As SQ index does not include any user information, the list of objects in each quadtree node is used as a whole and fully scanned for filtering objects that are posted by L_f users.

SSTkQ in TCSSQ index. In TCSSQ index, SSTkQ is processed using the same two phases that are used in SSQ index, with an extended user filtering step. As TCSSQ index maintains extra user list information $C.L_u$ in each quadtree node C, when a quadtree node is accessed, the user friend list $u.L_f$ is intersected with the node user list $C.L_u$. If the intersection is empty, then C and all its descendants are immediately pruned. Otherwise, C is considered for further processing as described in the two phases of SSQ index.

6 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of geo-social real-time indexing and query processing of SSTRQ and SSTkQ queries as discussed in previous sections. Section 6.1 explains the experimental settings. Sections 6.2-6.4 evaluate indexing scalability, memory consumption, and query evaluation, respectively.

6.1 Experimental Setup

We evaluate the three indexes that are discussed in Section 4 for indexing scalability, storage overhead, and query processing. The proposed Spatial-Social Quadtree index is denoted as *SSQ*, the baseline Spatial Quadtree index is denoted as *SQ*, and the Tightly-Coupled Spatial-Social Quadtree index is denoted as *TCSSQ*, a modified version of [39] for real-time operations. Our parameters include quadtree node size, dataset size, query answer size k, query range, and the space-time weighting parameter α. Unless mentioned otherwise, the default node size is 2000, dataset size is 80 million objects, k is 100, query range is 50 km, α is 0.2, R_{Max} is 500 km, T_{Max} is one day, and buffer size is 500K entries. Our performance measures include index digestion rate (the average number of indexed objects per second), index memory footprint, and query latency. All experiments are based on Java 8 implementation and using an Intel Xeon(R) server with CPU E5-2637 v4 (3.50 GHz) and 128GB RAM running Ubuntu 16.04.

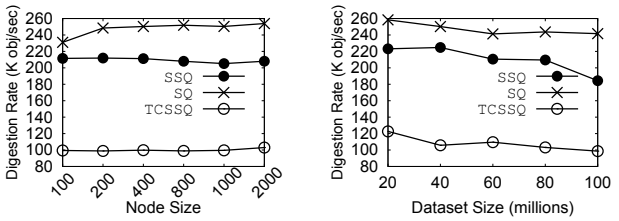
Evaluation datasets and query workloads. We have collected 6+ billion geotagged tweets from public Twitter Streaming APIs over the course of five years. Then, five datasets, of sizes 20, 40, 60, 80, and 100 million tweets, are composed for our evaluation. Each Tweet is represented with a latitude/longitude coordinates that represent either an exact location or a centroid of a place, e.g., a city or a landmark. Users of all tweets have been extracted from each of the five datasets. The data includes only the number of friends of each user and not the actual friend list. Thus, we randomly generate a list of friends for each user, where the majority are close to her location while the rest are scattered around the world. Table 2 summarizes the number of users and the average number of friends in each dataset. In order to generate the query workload, we randomly select a thousand users, and their home locations are the query points.

Dataset	20M	40M	60M	80M	100M
Users	3379403	4589750	5323808	5862339	6319263
Avg. Friends	531	513	504	497	492

Table 2: Evaluation Dataset Statistics

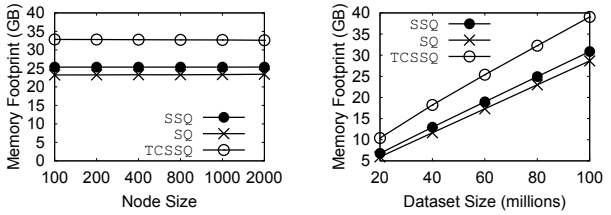
6.2 Indexing Scalability

This section evaluates the scalability of the real-time indexing measured as the number of objects being digested in a second. Figure 4a shows the indexing scalability with different quadtree node size. *SQ* can digest on average 250K objects/sec which is the highest among the three indexes. *SSQ* digestion rate is reduced to 210K objects/sec, due to incorporating social information in the index structure, which still maintains 84% of *SQ* digestion rate and digests an order of magnitude higher than Twitter rate. On the other hand, *TCSSQ* has the lowest digestion rate of 100K objects/sec due to



(a) Varying Node Sizes (b) Varying Dataset Sizes

Figure 4: Indexing Scalability



(a) Varying Node Sizes (b) Varying Dataset Sizes

Figure 5: Memory Footprint

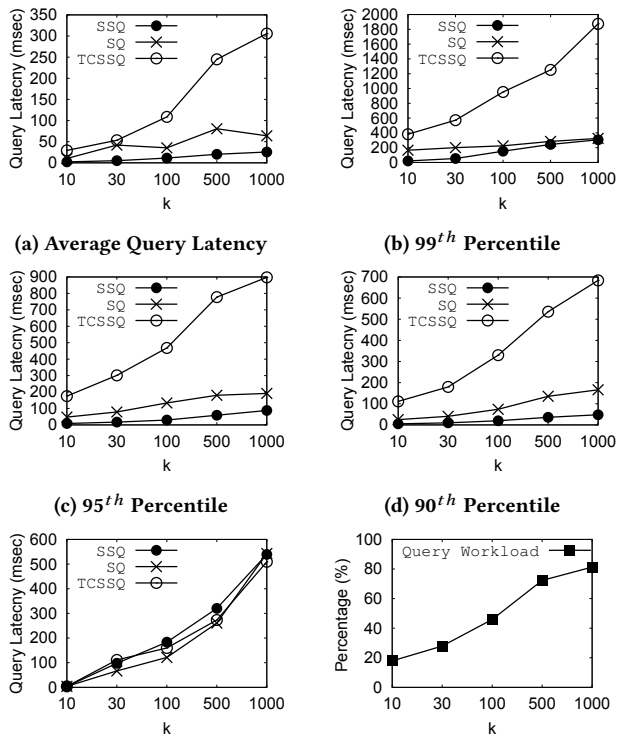
the overhead of summarizing all sub-tree social information. It is though noticeable that different node sizes have no real impact on the digestion rate.

Figure 4b shows the impact of different dataset sizes on the digestion rate. The digestion rate is slightly decreasing when the number of objects increases for all indexes due to the larger index contents, which makes it heavier to digest new data. However, the overall reduction is still acceptable. For example, *SSQ* digests 220K objects/sec with 20 millions objects and 190K objects/sec with 100 millions objects, which represents 14% reduction of digestion rate and both are still an order of magnitude higher than Twitter rate.

6.3 Memory Consumption

Figure 5 shows the memory consumption for the three indexes with varying the quadtree node size (Figure 5a) and varying dataset size (Figure 5b). Varying node size in Figure 5a does not significantly affect the memory consumption for all the three indexes despite an order of magnitude higher node capacity, which leads to significantly less number of index nodes. This shows the minor effect of the index nodes' memory on storage overhead as the majority of memory consumed for data that is being stored inside the nodes. *SQ* consumes the lowest memory, 22 GB, while *SSQ* consumes a slightly higher memory resource, 24 GB, since the index structure keeps more information about the social aspect. *TCSSQ* consumes the highest memory resource, 33 GB, with different index node sizes. The additional social information of *TCSSQ* index structure increases the memory overhead by ~50% of the baseline *SQ* index.

Varying the dataset size in Figure 5b affects the memory resources to be increased linearly for all alternatives. For example, *SSQ* consumes 7 GB when the dataset size is 20 million objects, and when the dataset size triple, *SSQ* consumes 19 GB. The same pattern repeats for *SQ* and *TCSSQ*, where always *TCSSQ* still consumes the largest memory. This also confirms that the majority of the



(a) Average Query Latency (b) 99^{th} Percentile

(c) 95^{th} Percentile (d) 90^{th} Percentile

(e) Disk Overhead (f) Social Expansion

Figure 6: Range query latency with varying k

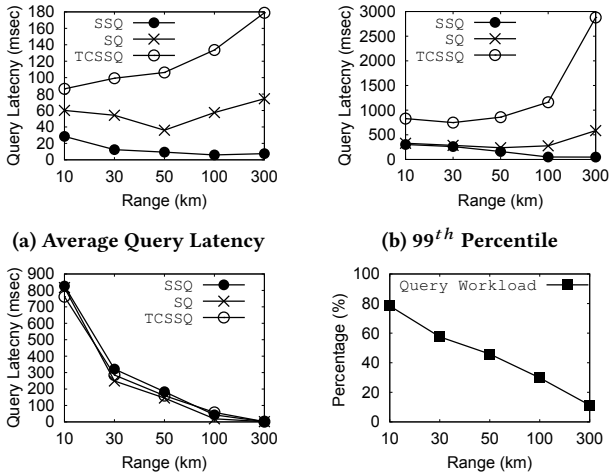
memory resources are being consumed by the data that resides in the main-memory.

6.4 Query Evaluation

This section evaluates the query processing of the Spatial-Social Temporal Range Query (SSTRQ) and the Spatial-Social Temporal kNN Query (SSTkQ), called for short range query and kNN query, respectively. The query latency is presented as an average and percentiles, e.g., the 99% percentile latency that shows the maximum query latency for 99% of the queries.

(a) SSTRQ Query Evaluation:

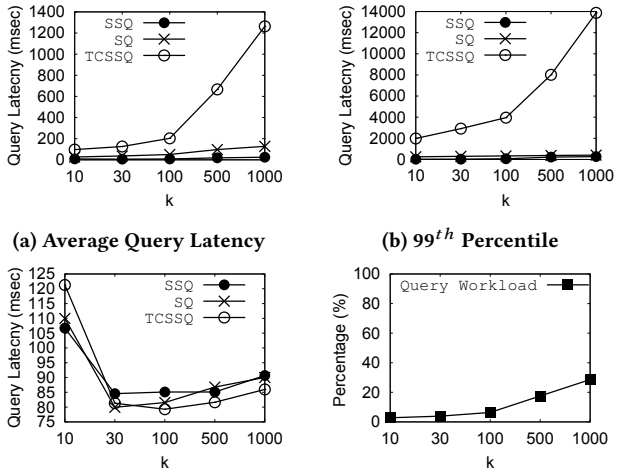
Effect of varying k. Figure 6 shows the effect of varying k on range query latency, both in-memory and disk processing. Figure 6a shows in-memory range query latency measured in milliseconds (msec) for all alternatives. Generally, query latency is increasing with increasing k due to the more processing needed for getting larger answer. However, the latency of *TCSSQ* is significantly higher than the other two alternatives. This overhead is caused by checking the internal nodes to prune some tree branches that are not promising to find the k objects. Although this process is effective in disk-based processing of traditional queries, in streaming environments, this process increases the real-time overhead tremendously. As shown in the figure, our proposed *SSQ* index performs the best with 2 msec latency at k=10, and it is increasing to 25 msec at k=1000. *SSQ* index combines both social-aware pruning and lightweight structure that is suitable for real-time environments. *SQ* index has no social awareness, so it is three times



(a) Average Query Latency (b) 99^{th} Percentile

(c) Disk Overhead (d) Social Expansion

Figure 7: Range query latency with varying range query



(a) Average Query Latency (b) 99^{th} Percentile

(c) Disk Overhead (d) Social Expansion

Figure 8: kNN latency with varying k

slower than *SSQ* index on average. It starts with 10 msec latency at k=10, and it is increasing steadily to reach 65 msec at k=1000. The superiority of *SSQ* index is further confirmed by measuring the 99th, 95th, 90th percentile latency as depicted in Figures 6b, 6c, and 6d, respectively. *SSQ* constantly performs the best in terms of query latency, and the advantage is even obvious in Figures 6c and 6d.

Figure 6e shows the disk overhead to retrieve the users' friends or friends of friends in order to retrieve the k objects for the given user. All indexes need to access the disk to fetch the social data. Therefore, all alternatives perform similarly, with increasing latency with larger k value, as all indexes use the same disk-based social structure. The increase with k value is explained by the percentage of the query being expanded beyond the first social level (direct friends) as shown in Figure 6f. The larger k, the less probability that direct friends can satisfy the query answer, and hence expansion to higher social levels is necessary.

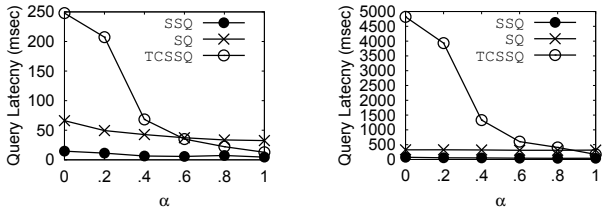
Effect of varying query range. Figure 7a shows the average query latency with varying query range from 10 km to 300 km. *SSQ* index still performs the best among the other alternatives. Both *SQ* and *TCSSQ* indexes have an increasing latency with the increasing range due to the larger search space. On the contrary, the query latency of *SSQ* drops with increasing range. As *SSQ* employs both temporal and social pruning; the more cells the more recent initial answer, which in turn produces a tight temporal upper bound. The temporal pruning uses this tight bound to terminate processing very early in many cells. In addition, the social pruning enables to process only the posting lists that are socially connected to the query issuer, which prunes a significant number of objects that do not contribute to the answer. At 10 km range, *SSQ* processes queries with an average of 27 msec latency, while at the range of 300 km, this latency drops four times to 7 msec. On another hand, *SQ* has almost a stable performance with varying ranges as it only employs the temporal pruning, while *TCSSQ* performs the worst despite it employs both the temporal and social pruning for the same reasons that are discussed before. Figure 7b shows the 99th percentile latency, which confirms the superiority of *SSQ* over all alternatives.

Figures 7c and 7d show the correlation between disk overhead and the percentage of queries being expanded to higher social levels. Clearly, the disk overhead decreases when the expansion percentage decreases. With small spatial ranges, the probability to retrieve k objects from direct friends is small, and hence the majority of queries expand. This significantly decreases with increasing range.

(b) SSTkQ Query Evaluation:

Effect of varying k. Figure 8a shows the in-memory query latency with varying k. *SSQ* index performs consistently better than the other alternatives due to its three-dimensional pruning on temporal, spatial, and social dimensions. At k=10, *SSQ* has an average query latency of 9 msec, which increases with larger k to 25 msec at k=1000. This is fifty times better than *TCSSQ* due to its social pruning overhead that is not suitable for real-time processing. On the contrary, *SQ* is slower three times compared to *SSQ* due to lack of social pruning. Such behavior remains the same for the 99th percentile of queries, as shown in Figure 8b, which shows the superiority of *SSQ* in all cases. For disk overhead, all alternative incur almost the same latency as shown in Figure 8c due to using the same disk structure. Also, the percentage of socially expanded kNN queries, depicted in Figure 8d, are much less than range queries since range queries are restricted by a spatial range, which obligates to expand the search to higher social levels often.

Effect of varying α. Figure 9a shows the effect of varying α that controls the relative importance of the spatial and temporal scores in the spatio-temporal distance. As the figure shows, the α value has a great impact on the query performance, especially for *TCSSQ* index. When only the temporal score is important (at α=0), all indexes hit their highest query latency because the query processor has to cover a larger search region. With increasing α, the query latency gradually drops to the lowest point for all the indexes when only the spatial score is important (at α=1). For all values of α, *SSQ* performs the best, while *TCSSQ* performs the



(a) Average Query Latency **(b) 99^{th} Percentile**

Figure 9: kNN latency with varying α

worst up to $\alpha < 0.6$. Then, *TCSSQ* performs better than *SQ* after $\alpha \geq 0.6$. The key reason behind this behavior is the number of cells that need to be processed is huge with small α, and *TCSSQ* is very sensitive to the number of cells as it checks for overlap with long user lists. This number decreases as the query region shrinks due to the importance shifts to the spatial closeness. Figure 9b confirms similar behavior and *SSQ* superiority on the 99th percentile of queries. For different values of α, the disk overhead is almost stable (approximately 80 msec) for all alternatives except with α=0 where very few queries expand the search space, which makes the disk overhead very minimal with a few milliseconds.

7 CONCLUSION

This paper defined two temporal geo-social queries on streaming data as extensions for the fundamental spatial k-nearest neighbor (kNN) and range queries. To address these queries, we proposed a generic indexing framework for real-time geo-social data that digests and indexes highly-dynamic data in main-memory and organizes stable social information in a disk-based structure. Based on this framework, we proposed spatial-social quadtree index that is lightweight to handle real-time data efficiently, while providing scalable query response for both kNN and range queries. In addition, we adopted two baseline index structures based on the proposed indexing framework. The experimental evaluation on real datasets has clearly shown the superiority of our proposed index for both real-time indexing and query processing.

REFERENCES

[1] H. Abdelhaq, C. Sengstock, and M. Gertz. Eventweet: Online Localized Event Detection from Twitter. *VLDB*, 2013.
[2] R. Ahuja, N. Armenatzoglou, D. Papadias, and G. J. Fakas. Geo-social Keyword Search. In *SSTD*, 2015.
[3] A. Almaslukh and A. Magdy. Evaluating Spatial-keyword Queries on Streaming Data. In *SIGSPATIAL*, 2018.
[4] W. G. Aref and H. Samet. Efficient Processing of Window Queries in the Pyramid Data Structure. In *SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1990.
[5] N. Armenatzoglou, R. Ahuja, and D. Papadias. Geo-social Ranking: Functions and Query Processing. *VLDB Journal*, 2015.
[6] N. Armenatzoglou, S. Papadopoulos, and D. Papadias. A General Gramework for Geo-social Query Processing. *VLDB*, 2013.
[7] J. Bao, M. F. Mokbel, and C.-Y. Chow. Geofeed: A Location Aware News Feed System. In *ICDE*, 2012.
[8] J. Bao, Y. Zheng, and M. F. Mokbel. Location-based and Preference-aware Recommendation using Sparse Geo-social Networking Data. In *GIS*, 2012.
[9] C. Budak, T. Georgiou, D. Agrawal, and A. El Abbadi. Geoscope: Online Detection of Geo-correlated Information Trends in Social Networks. *VLDB*, 2013.
[10] J. Chae, D. Thom, H. Bosch, Y. Jang, R. Maciejewski, D. S. Ebert, and T. Ertl. Spatiotemporal Social Media Analytics for Abnormal Event Detection and Examination Using Seasonal-trend Decomposition. In *IEEE VAST*, 2012.
[11] L. Chen, G. Cong, X. Cao, and K.-L. Tan. Temporal Spatial-keyword Top-k Publish/Subscribe. In *ICDE*, 2015.
[12] G. Cong, C. S. Jensen, and D. Wu. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *VLDB*, 2009.
[13] T. Emrich, M. Franzke, N. Mamoulis, M. Renz, and A. Züfle. Geo-social Skyline Queries. In *DASFAA*, 2014.
[14] The Top 20 Valuable Facebook Statistics. https://zephoria.com/top-15-valuable-facebook-statistics/, 2019. May 2019.
[15] W. Feng, C. Zhang, W. Zhang, J. Han, J. Wang, C. Aggarwal, and J. Huang. STREAMCUBE: Hierarchical Spatio-temporal Hashtag Clustering for Event Exploration over the Twitter Stream. In *ICDE*, 2015.
[16] L. Hong, A. Ahmed, S. Gurumurthy, A. J. Smola, and K. Tsioutsiouliklis. Discovering Geographical Topics in the Twitter Stream. In *WWW*, 2012.
[17] D. Hristova, M. J. Williams, M. Musolesi, P. Panzarasa, and C. Mascolo. Measuring Urban Social Diversity Using Interconnected Geo-social Networks. In *WWW*, 2016.
[18] Q. Huang and Y. Liu. On Geo-social Network Services. In *2009 17th International Conference on Geoinformatics*, 2009.
[19] Internet Live Stats 2019. http://internetlivestats.com/, 2019. May 2019.
[20] J. Jiang, H. Lu, B. Yang, and B. Cui. Finding Top-k Local Users in Geo-tagged Social Media Data. In *ICDE*, 2015.
[21] R. Lee and K. Sumiya. Measuring Geographical Regularities of Crowd Behaviors for Twitter-based Geo-social Event Detection. In *SIGSPATIAL LSBN Workshop*, 2010.
[22] Y. Li, Z. Bao, G. Li, and K.-L. Tan. Real Time Personalized Search on Social Networks. In *ICDE*, 2015.
[23] Y. Li, R. Chen, J. Xu, Q. Huang, H. Hu, and B. Choi. Geo-social k-cover Group Queries for Collaborative Spatial Computing. *TKDE*, 2015.
[24] W. Liu, W. Sun, C. Chen, Y. Huang, Y. Jing, and K. Chen. Circle of Friend Query in Geo-social Networks. In *DASFAA*, 2012.
[25] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xing. Discovering Spatio-temporal Causal Interactions in Traffic Data Streams. In *SIGKDD*, 2011.
[26] A. Magdy, L. Alarabi, S. Al-Harthi, M. Musleh, T. M. Ghanem, S. Ghani, and M. F. Mokbel. Taghreed: A System for Querying, Analyzing, and Visualizing Geotagged Microblogs. In *SIGSPATIAL*, 2014.
[27] A. Magdy, R. Alghamdi, and M. F. Mokbel. On Main-memory Flushing in Microblogs Data Management Systems. In *ICDE*, 2016.
[28] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He. Mercury: A Memory-constrained Spatio-temporal Real-time Search on Microblogs. In *ICDE*, 2014.
[29] A. R. Mahmood, A. M. Aly, and W. G. Aref. FAST: Frequency-Aware Indexing for Spatio-Textual Data Streams. In *ICDE*, 2018.
[30] S. Nishio, D. Amagata, and T. Hara. Geo-Social Keyword Top-k Data Monitoring over Sliding Window. In *DEXA*, 2017.
[31] T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake Shakes Twitter Users: Real-time Event Detection by Social Sensors. In *WWW*, 2010.
[32] J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling. TwitterStand: News in Tweets. In *SIGSPATIAL*, 2009.
[33] A. Sohail, M. A. Cheema, and D. Taniar. Social-Aware Spatial Top-k and Skyline Queries. *The Computer Journal*, 2018.
[34] A. Sohail, G. Murtaza, and D. Taniar. Retrieving Top-k Famous Places in Location-based Social Networks. In *Australasian Database Conference*, 2016.
[35] P. Symeonidis, A. Papadimitriou, Y. Manolopoulos, P. Senkul, and I. Toroslu. Geo-social Recommendations Based on Incremental Tensor Reduction and Local Path Traversal. In *SIGSPATIAL International Workshop on Location-Based Social Networks*, 2011.
[36] Twitter by the Numbers: Stats, Demographics & Fun Facts. https://www.omnicoreagency.com/twitter-statistics/, 2019.
[37] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. Ap-tree: Efficiently Support Continuous Spatial-keyword Queries over Stream. In *ICDE*, 2015.
[38] H. Wei, J. Sankaranarayanan, and H. Samet. Enhancing Local Live Tweet Stream to Detect News. In *SIGSPATIAL LENS Workshop*, 2018.
[39] D. Wu, Y. Li, B. Choi, and J. Xu. Social-aware Top-k Spatial Keyword Search. In *MDM*, 2014.
[40] D.-N. Yang, C.-Y. Shen, W.-C. Lee, and M.-S. Chen. On Socio-spatial Group Query for Location-based Social Networks. In *SIGKDD*, 2012.
[41] H. Yin, Z. Hu, X. Zhou, H. Wang, K. Zheng, Q. V. H. Nguyen, and S. Sadiq. Discovering Interpretable Geo-social Communities for User Behavior Prediction. In *ICDE*, 2016.
[42] Q. Yuan, G. Cong, Z. Ma, A. Sun, and N. M. Thalmann. Time-aware Point-of-Interest recommendation. In *SIGIR*, 2013.
[43] J.-D. Zhang and C.-Y. Chow. iGSLR: Personalized Geo-social Location Recommendation: a Kernel Density Estimation Approach. In *SIGSPATIAL*, 2013.
[44] J.-D. Zhang and C.-Y. Chow. GeoSoCa: Exploiting Geographical, Social and Categorical Correlations for Point-of-Interest Recommendations. In *SIGIR*, 2015.
[45] J. Zhao, Y. Gao, G. Chen, C. S. Jensen, R. Chen, and D. Cai. Reverse Top-k Geo-social Keyword Queries in Road Networks. In *ICDE*, 2017.